Multi-Agent Deep Reinforcement Learning Algorithms

by

Kenny Song

New York University Shanghai

Computer Science Department

May, 2017


上海纽约大学
NYU SHANGHAI

Acknowledgments

A thank you to Keith Ross and Martin Arjovsky for their guidance, wisdom, and debugging help on this project. This has been a great learning experience, and has motivated me to continue my machine learning studies in the future.

Contents

Introduction

Deep reinforcement learning (DRL) combines reinforcement learning algorithms with (deep) neural networks as function approximators. Impressive progress has been made in this domain in the recent few years, notably using DRL to surpass human-level performance on Atari video games with only pixel data[5] and to beat Go grandmasters[8].

Most of the current research and applications are based on single-agent tasks, e.g. single player games such as Pacman. The standard reinforcement learning algorithms, such as policy gradient or Q-learning, do not scale to multi-agent tasks as the action spaces can be very high-dimensional; for n agents, each with m actions denoted as one-hot vectors, the joint action space is $\{0, 1\}^{m^n}$. With traditional approaches, the neural network needed to map this space would be impractically large.

Our goal was to develop multi-agent algorithms where the network size remains constant even as the number of agents grows. This would allow the network to effectively learn structure in this action space and discover optimal coordinated action between agents. We explored two different algorithms:

1. Policy gradient with an LSTM policy net and MLP state value baseline
2. SARSA approach with a hybrid LSTM-MLP Q-function

These have generated promising early results by learning better multi-agent policies than the baseline for several tasks, and have also solved environments that were too large for traditional approaches to tackle. However, more work remains to fine-tune these algorithms and explore alternative approaches.

Background & Related Work

Standard reinforcement learning algorithms include value-function based approaches and policy-function based approaches (or both, called actor-critic)[7]. Value-based algorithms are perhaps the simplest approach, where a state or state-action function is learned. However, value functions are also indirect and not scalable, as agents need to maximize the values over a potentially large action space to follow an ϵ-greedy policy.

In contrast, policy-based approaches work to directly learn a policy function, which gives action probabilities at a specific state, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(s, \cdot)$ is a probability distribution over $\mathcal{A}$. The state space is denoted by $\mathcal{S}$, and the action space by $\mathcal{A}$. In deep reinforcement learning, π is approximated with a neural network and trained with policy-based algorithms. A convenient notation is $\pi_\theta(a \mid s)$, which reads as the probability of selecting action a given state s, where the policy network has parameters θ.

The canonical policy-based method is REINFORCE, also called policy gradient, where we apply stochastic gradient ascent on $J(\theta)$, a performance metric of the policy π_θ usually interpreted as the value of the start state[14]. Subtracting a baseline function, usually state-values, can reduce the variance of gradient updates and thus increase the speed of learning. Later, this theory was extended to use function approximators for the policy and baseline functions, which applies to neural network based approaches as well[12]. A recent modification called Asynchronous Advantage Actor-Critic can further boost performance[4].

Most recent successes in deep reinforcement learning involve using deep neural networks to approximate value functions. A notable example is using deep Q-networks to surpass human-level performance on Atari video games[5]. Deep policy gradient has been successful at playing Pong[3], but has not been systematically applied to more complex games. In particular, deep networks seem to be effective at learning from raw pixel data.

Thus, most of deep reinforcement learning has focused on approximating value functions for single agent tasks, since multi-agent joint action spaces grow exponentially more costly to maximize over. A recent paper[9] focused on teaching individual agents to communicate in order to find multi-agent policies, which is a different approach to this problem. We do not explore their paper in depth here, though it may serve as a state-of-the-art comparison in the future. There have been no other significant attempts at applying deep reinforcement learning to multiple agents.

Policy Gradient

Single-Agent Policy Gradient

Conceptually, a policy is a plan that tells the agent which action to take at a given state, usually given as action probabilities. Thus, a policy is the function, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(\cdot \mid s) := \pi(s, \cdot)$ is a probability distribution over $\mathcal{A}$. This policy can be modeled with a neural network, written as π_θ, where θ represents the weights of the network.

To be precise, the neural network (MLP) would have an input layer of size $|\mathcal{S}|$ and an output layer of size $|\mathcal{A}|$, with an arbitrary number of intermediary hidden layers*. The output layer will contain a softmax, which normalizes the output values to valid probabilities. That is, $\text{MLP}_\theta : \mathcal{S} \to [0, 1]^{|\mathcal{A}|}$, which we can index into to get values of $\pi_\theta(a \mid s)$.

How may we judge how good a policy is? One metric, which works for both episodic and continuing environments, is the start value. This is the discounted return we would expect to get from the start state by following a given policy,

$$J(\theta) := v_{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right].$$

In traditional neural network terms, $J(\theta)$ is simply the objective function of our policy network. We do not actually need to separately model the value function $v(s)$, as this objective function is never evaluated, but implicit in the policy gradient algorithm (if a state value baseline is used, however, we would need to model v_{π_θ}).

At this point, we can basically do some variant of gradient ascent on $J(\theta)$ to maximize the "goodness" of our policy and converge to the optimal policy. This is the idea of the REINFORCE algorithm[14] for policy-based learning, also just called policy gradient.

The Policy Gradient Theorem defines the gradient of our objective function as[12],

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) q_{\pi_\theta}(s, a)].$$

In practice, we estimate the unknown q function with empirical Monte-Carlo returns,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) G_s].$$

We also subtract a baseline function $b(s)$ from G, which reduces the variance of the algorithm and leads to faster learning[12]. Note that the baseline is only a function of the state. This is usually turns out to be the state-value function $v(s)$. The update then becomes,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a)(G_s - v(s))].$$

*In practice, we may not use one-hot state vectors, so the input layer would have size $< |\mathcal{S}|$, but we stick to this assumption in this paper, wherever $|\mathcal{S}|$ is mentioned. Actions are generally one-hot vectors, though.

Last, we generally just sample from the expectation for stochastic gradient ascent,

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(s, a)(G_s - v(s)).$$

Thus, the weight update can be done for each step in an episode, and has the form,

$$\Delta\theta = \alpha \nabla_\theta J(\theta) = \alpha \underbrace{\nabla_\theta \log \pi_\theta(s, a)}_{\text{score function}} (G_s - v(s)).$$

This score function is calculated with standard backpropagation through the MLP. The baseline function $v(s)$ is modeled by a separate MLP, and is trained separately using episode returns, with standard supervised learning techniques.

Multi-Agent Policy Gradient

Recall that the MLP policy network has an input layer of size $|\mathcal{S}|$ and an output layer of size $|\mathcal{A}|$. For multi-agent tasks, $|\mathcal{A}|$ is exponential in the number of agents. Suppose each agent has m actions, that is, its actions are one-hot vectors in $\{0, 1\}^m$. If there are n agents, then their joint actions are one-hot vectors in $\{0, 1\}^{m^n}$. Thus, the output layer of the MLP policy network would be of size m^n. We reach practical limits of this strategy very quickly; for only 6 agents with 9 actions each (movements on a board), $|\mathcal{A}| = 9^6 = 531,441$.

Clearly, a better approach is needed to deal with large action spaces for multiple agents. The policy is essentially trying to learn the massive joint probability distribution,

$$\pi(a \mid s) = \mathbb{P}[A = a \mid S = s] = \mathbb{P}[A_1 = a_1, A_2 = a_2, \ldots, A_n = a_n \mid S = s],$$

where A_i is the action of agent i. Using the chain rule of probability, we can factor this joint distribution into a product of conditional, one-dimensional distributions,

$$\mathbb{P}[A_1, A_2, \ldots, A_n \mid S] = \mathbb{P}[A_1 \mid S] \times \mathbb{P}[A_2 \mid S, A_1] \times \cdots \times \mathbb{P}[A_n \mid S, A_1, \ldots, A_{n-1}].$$

It turns out that recurrent neural networks, LSTMs in particular, are perfect for the task of learning these conditional distributions. This technique was recently applied in neural machine translation to learn an LSTM language model[10]. In general, any recurrent neural network will have the form,

$$h_d = f_\theta(x_d, h_{d-1}),$$

where x_d is the input at time-step d and h_d is the hidden state of the network at time-step d. An LSTM, of course, has an additional cell state term, C, that is transferred between time-steps, so we may view it as,

$$h_d, C_d = f_\theta(x_d, h_{d-1}, C_{d-1})$$

The additional parameters of an LSTM (forget, input, output gates) are summarized in the θ term. For a more extensive treatment of LSTMs, see the Deep Learning Book[2].

Now, let the LSTM f_θ at time-step d have input $x_d = \texttt{concat}(s, a_{d-1})$. Given this input, as well as information from previous time-steps encoded in h_{d-1} and C_{d-1}, the LSTM at time-step d therefore corresponds to the conditional distribution $\mathbb{P}[A_d \mid S, A_1, \ldots, A_{d-1}]$ for agent d. (For $d = 1$, we initialized inputs C_0, h_0, and a_0 to zero.) This is our LSTM policy network, which at each time-step provides the action probabilities for one agent, conditioned on the state and actions of previous agents, and is run for n time-steps.

This LSTM has input layer size $|\mathcal{S}| + |\mathcal{A}_i|$ and output layer size $|\mathcal{A}_i|$, where $\mathcal{A}_i$ is the action space for agent i. Thus, the network size remains constant as the number of agents grows, scaling perfectly for multi-agent tasks. There is the consideration of what to do if the agents have different action spaces; one approach would be to simply set the output layer to size $|\bigcup_i \mathcal{A}_i|$ and perform rejection sampling to select actions. For our tasks, we assumed equivalent agent action spaces, but dealt with issue of state-dependent action spaces with a masked softmax function, see Evaluation.

An important distinction: the time-steps of the LSTM and time-steps of the episode are separate concepts. For each time-step t in the episode, $t = 1, \ldots, T$, the LSTM policy network runs for n time-steps, computing the action probabilities for each agent $d = 1, \ldots, n$.

For the LSTM policy net, we can factor the score function into a sum of conditional distributions, which we can efficiently backpropagate through,

$$\begin{aligned}\nabla_\theta \log \pi_\theta(s, a) &= \nabla_\theta \log \prod_{d=1}^{n} \mathbb{P}_\theta[a_d \mid a_1, \ldots, a_{d-1}, s] \\ &= \nabla_\theta \sum_{d=1}^{n} \log \mathbb{P}_\theta[a_d \mid a_1, \ldots, a_{d-1}, s].\end{aligned}$$

SARSA

Single-Agent SARSA

While policy gradient is a policy-based method, meaning that it directly learns a policy function, SARSA is a value-based method that learns a value function[11]. One value function we've already seen is the state-value function, $v(s)$, which gives the value of being in a certain state. SARSA tries to learn the $q(s, a)$ value function, which gives the value of taking a specific action from a state.

Value functions induce a policy under the principle that higher-value actions should be taken more often. The simplest induced policy is then a greedy policy, where the action $\text{argmax}_a\, q(s, a)$ is always chosen. An ϵ-greedy policy behaves in the same way, except for selecting a random action with probability ϵ, which encourages exploration. A softmax policy involves normalizing the action-values through a softmax function, and then sampling from this distribution.

Traditionally, the q function is given in tabular form, that is, it is just a lookup table with an entry for every (s, a) pair. The SARSA algorithm was designed for this approach, and has the following steps,

1. At state s_t, sample a_t from $q(\cdot, s_t)$ (with ϵ-greedy, softmax, etc)
2. Take the action a_t, and record r_{t+1}, s_{t+1} from the environment
3. At state s_{t+1}, sample a_{t+1} from $q(\cdot, s_{t+1})$ without taking the action
4. Update the $q(s_t, a_t)$ entry towards $r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ (weighted average)
5. Repeat from (1) until convergence

However, tabular forms are completely non-scalable, so q functions are generally modeled with a MLP, denoted q_θ. The input layer has size $|\mathcal{S}| + |\mathcal{A}|$ with a scalar output layer. The update (step 4 above), must then be updated to work with an MLP. Since the idea is to move $q(s_t, a_t)$ towards $r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$, we can use gradient updates to minimize the squared error between these terms,

$$\Delta\theta = -\alpha \nabla_\theta \Big[\underbrace{r_{t+1} + \gamma q_\theta(s_{t+1}, a_{t+1})}_{\text{treated as a constant term}} - q_\theta(s_t, a_t) \Big]^2.$$

Note that the underlined target term is treated as constant, so its gradient is zero. We can simplify the weight update to:

$$\Delta\theta = \alpha(r_{t+1} + \gamma q_\theta(s_{t+1}, a_{t+1}) - q_\theta(s_t, a_t))\nabla_\theta q_\theta(s_t, a_t).$$

Multi-Agent SARSA

Using the MLP architecture above again causes scaling issues with a large action space, since the input layer is of size $|\mathcal{S}| + |\mathcal{A}|$. Thus, we need a q function that scales independently of the action space size. An initial approach may be to just use our LSTM policy net again, where we define q_θ as,

$$q_\theta(s, a) := \log \pi_\theta[A = a \mid S = s] = \sum_{d=1}^{n} \log \mathbb{P}_\theta[a_d \mid a_1, \ldots, a_{d-1}, s].$$

This makes sense as log is monotonically increasing, so q_θ assigns higher values to actions that higher probabilities. It also has the nice property that a softmax policy over this value function gives us back the LSTM policy probabilities:

$$\text{softmax}_a(q_\theta(s, \cdot)) = \frac{\exp(q_\theta(s, a))}{\sum_{a' \in \mathcal{A}} \exp(q_\theta(s, a'))} = \frac{\exp(\log \mathbb{P}_\theta[a \mid s])}{\sum_{a' \in \mathcal{A}} \exp(\log \mathbb{P}_\theta[a' \mid s])} = \mathbb{P}_\theta[a \mid s].$$

This solves the core scalability issue of value-based approaches, as we no longer need to maximize the q function over a large action space (as would be needed for ϵ-greedy). However, our q_θ network is somewhat limited in that it can only learn q functions satisfying the constraint,

$$\sum_{a \in \mathcal{A}} \exp(q(s, a)) = 1.$$

It is unclear if this poses any practical difficulties, but we would like to remove this constraint if possible. So, we added a log-partition function $k(s)$ to q_θ, which depends only on the state. Then, we have,

$$q_\theta(s, a) := \log \mathbb{P}_\theta[a \mid s] + k(s),$$
$$\sum_{a \in \mathcal{A}} \exp(q_\theta(s, a)) = \sum_{a \in \mathcal{A}} \exp(\log \mathbb{P}_\theta[a \mid s]) \exp(k(s)) = \exp(k(s)).$$

Since $k(s)$ is an arbitrary function, there is no more constraint on the sum of $\exp(q_\theta)$, and we can now model all possible q functions! The k function can be modeled by a separate MLP with an input layer of size $|\mathcal{S}|$ and scalar output layer. Thus, our q_θ is a hybrid LSTM-MLP system, where both networks are trained simultaneously, and the network size is independent of the number of agents. The gradient weight update term is exactly the same as for single-agent SARSA.

Implementation

These algorithms were implemented in PyTorch v0.1.12, a tensor and neural network library developed by Facebook AI Research. We originally experimented with using Keras with a Theano backend, but it lacked the flexibility needed to calculate gradients for our policy network. PyTorch is more imperative and transparent, which allowed us to more easily write and test these algorithms.

Evaluation and testing were performed with custom, multi-agent tasks written in vanilla Python 2.7 with Numpy, as we did not find suitable open source implementations of tasks for multiple agents.

We utilized the NYU Shanghai HPC system to train these models, and all code was written with GPU support. For the smaller tasks, there was little performance gain from running on GPU vs CPU. However, for more complex tasks (hunters with a large number of agents), GPUs significantly improved the absolute training speed.

At one point, all 2 TB of memory on an HPC node was exhausted by a long-running training script. I eventually traced this to a memory leak in PyTorch itself, which will be fixed in the next major release (0.2). This is one of the perils of using beta-stage software.

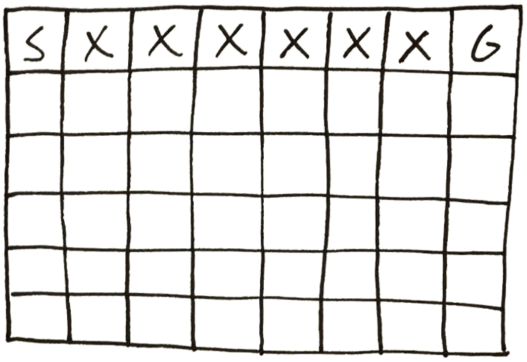
All code is available open source at: https://github.com/kennysong/deep_rl

Evaluation

Multi-Agent Tasks

We created two custom multi-agent tasks to evaluate our learning algorithms on, as no simple multi-agent tasks were readily available. The canonical multi-agent game, Starcraft, is too complex to be a reasonable starting point.

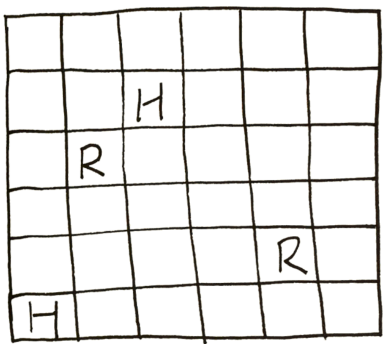
1. Gridworld



The first is a standard Gridworld task, adapted from the Cliff Walking task in the Sutton & Barto textbook[11]. On an $n \times m$ grid, two agents (one vertical and one horizontal) cooperate to control the position of a player. The goal is for the player to reach the fixed goal (top-right) from a fixed starting position (top-left), with a cliff in between. Actions including moving to any adjacent cell and staying in place. Every step that the player is not at the goal incurs a reward of -1, and falling into the cliff incurs a reward of -100 and a state reset back to the start. For efficient training, episodes that run over $10{,}000$ steps are cut off and given a penalty. There is no discounting.

This was also generalized to a 3-dimensional grid, where the cliff was the entire floor except for the start and goal, and floors above are empty cells. In this case, three agents cooperate to each control the x, y, and z positions. The reward structure is the same.

2. Hunters



The second is a Hunters vs. Rabbits task, where k hunters try to catch m rabbits on a square $n \times n$ grid. Each hunter corresponds to one agent, who solely controls the hunter's

position. Agents/hunters may have to cooperate to catch all rabbits. For example, in the grid setup above, the optimal strategy is not as simple as each hunter independently moving towards their closest rabbit. This task works well to test the scalability of our algorithms, as we can easily inflate the action space by adding additional hunters, without needing to alter the board as in 3-D Gridworld.

Capturing a rabbit will generate a reward of $+1$, and there is no penalty per time-step. However, a discount factor of $\gamma \in [0.7, 1)$ was chosen to incentivize hunters to complete the task more rapidly. Each episode is initialized with random positions for hunters and rabbits, rabbits are stationary within an episode, and both hunters and rabbits disappear on capture. Additional game settings include having rabbits move randomly or opposite to the nearest hunter, and keeping hunters in the game after capture.

*. State-Dependent Action Spaces

One consideration in our environments is that the action spaces are state-dependent. For example, when an agent is against the boundary of the grid, it cannot move into the wall. Thus, it was necessary to devise a way to restrict the action space based on the state.

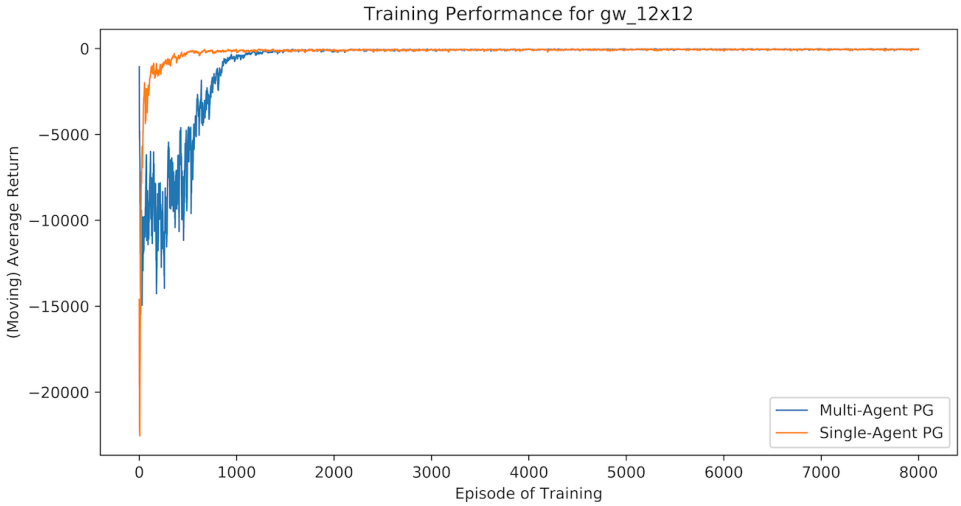
Our custom tasks provided a function to return a bitmask of the actions available to a certain agent at a given state. This bitmask was used in conjunction with a masked softmax function to generate filtered action probabilities. This softmax was made numerically stable by adapting the exp-normalize trick[13],

$$\pi_i(x, m) = \frac{e^{x_i - b} * m}{\sum_i e^{x_i - b} * m}, \text{ where } b = \max_{i \in m}\{x_i\}.$$

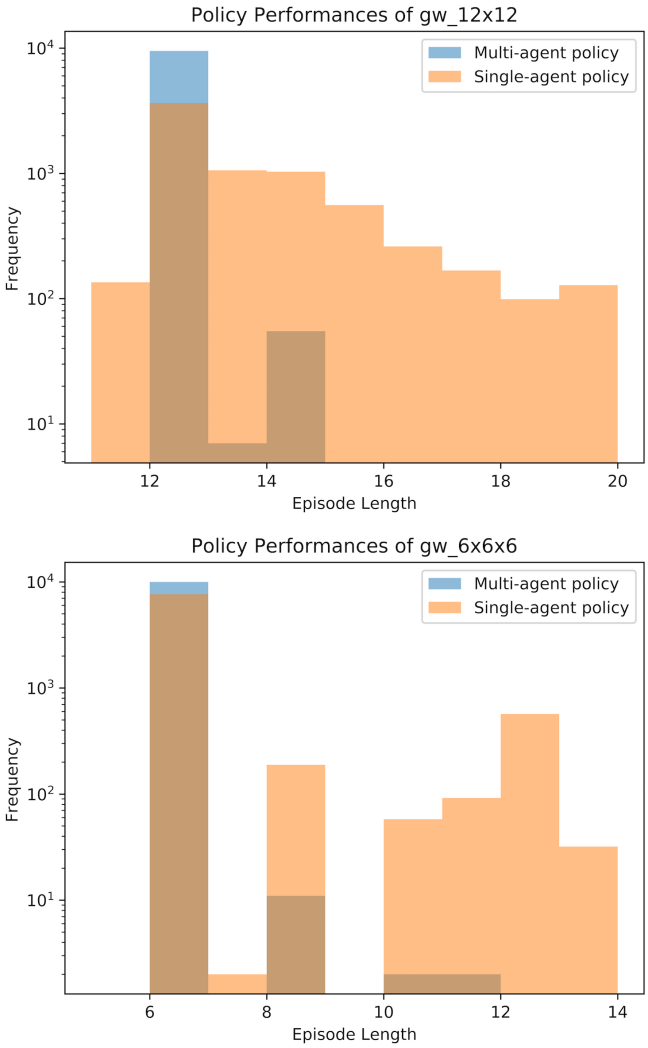
This masked softmax technique could also be used for agent-dependent action spaces.

Results

For the evaluation, we compared our multi-agent algorithms against standard single-agent algorithms as the baseline, except when the task was too large for the baseline to handle. First, we can look at the learning rate and policy performance on the Gridworld task with policy gradient.



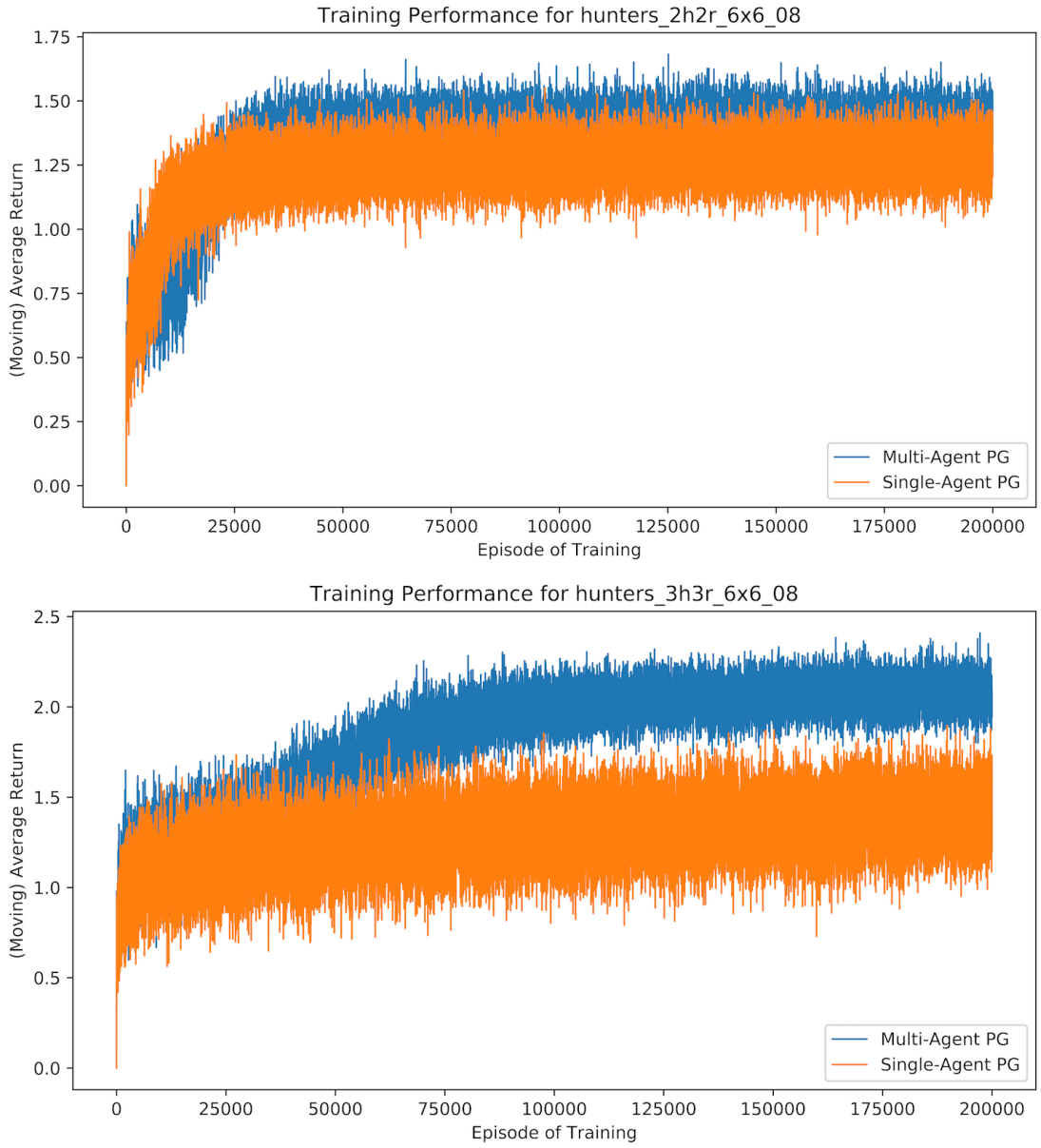
The top graph corresponds to a Gridworld of size 12×12, and the bottom corresponds to a Gridworld of size $6 \times 6 \times 6$. In both cases, we see that the single-agent MLP baseline converges slightly faster than the LSTM policy network. For the 3-D Gridworld, the baseline network is somewhat unstable for the first few thousand episodes, though this behavior is somewhat random and may also be stable for some runs. Thus, it is not conclusive that the LSTM policy network is more stable than the baseline.



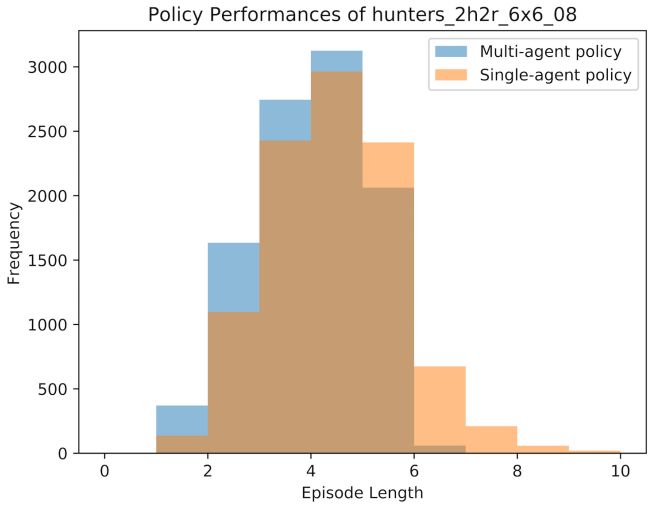
We can also directly evaluate the learned policies after 100,000 episodes of training on 10,000 new games. Note that these policies are almost, but not completely, deterministic (which is why a log histogram is used). Shorter episode lengths are better, as it indicates that the agents are finding the goal in less steps.

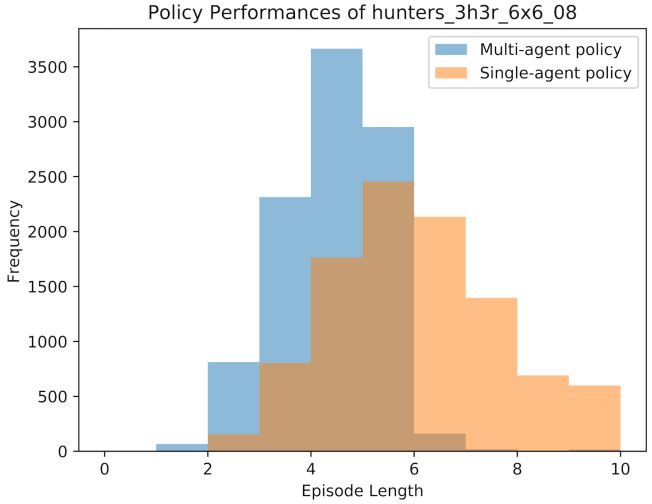
For the 12×12 grid, we can see that the single-agent baseline finds the optimal policy of 11 steps, while the LSTM policy only gets to a 12-step suboptimal policy. For the $6 \times 6 \times 6$ grid, neither approach finds the optimal 5-step policy, but the LSTM policy does choose better paths more frequently. This suggests that the multi-agent approaches may work better for more complex tasks.

Looking next at the Hunters task with 2 hunters + 2 rabbits and 3 hunters + 3 rabbits,



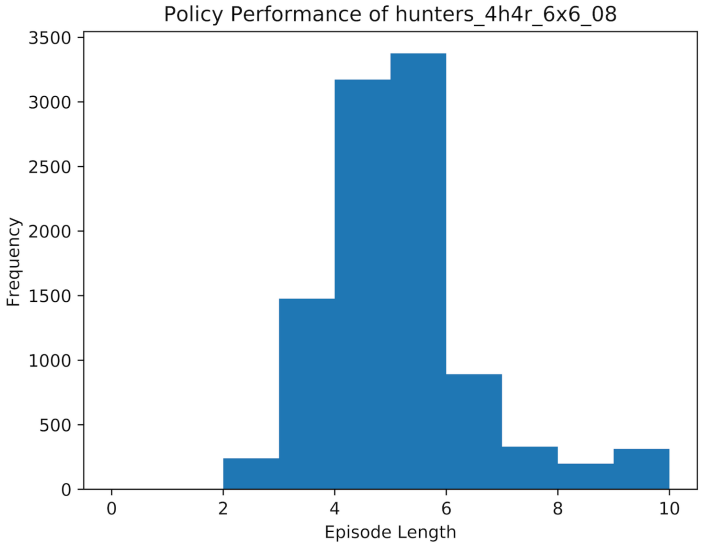
For 2 hunters, we can again see that the single-agent baseline learns slightly faster than the LSTM policy, but plateaus at a suboptimal policy. For 3 hunters, the baseline loses this edge and performs even more poorly compared to the LSTM policy.



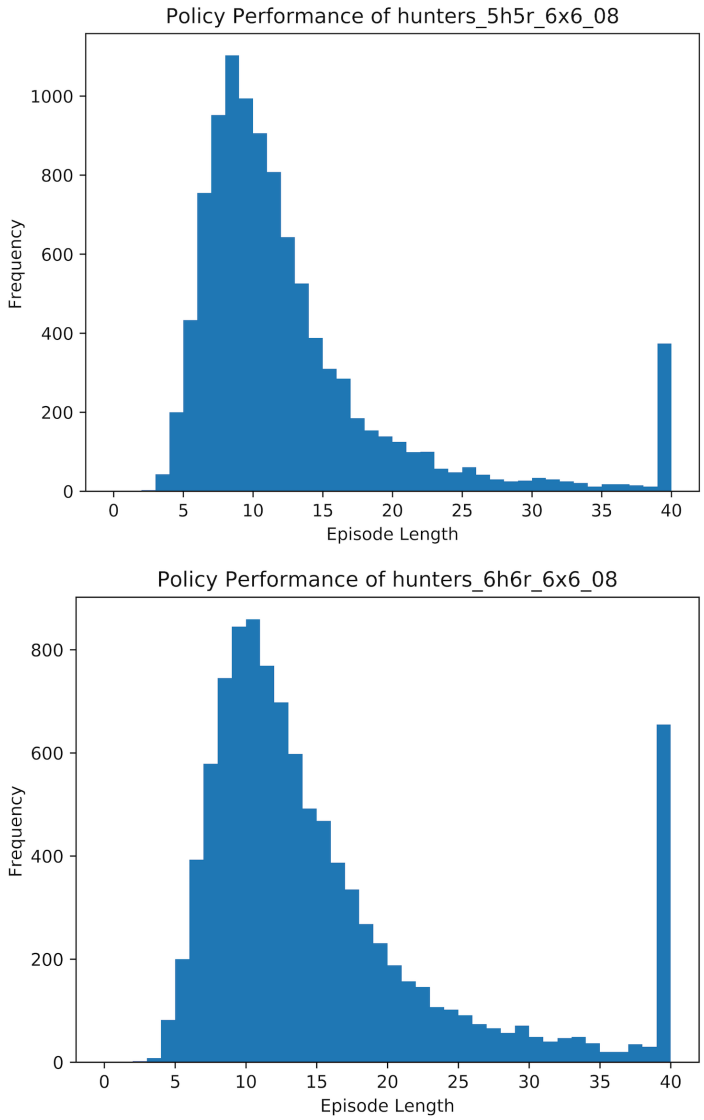


Now, directly evaluating the learned policies after $1{,}000{,}000$ episodes of training on $10{,}000$ new games with random starting states. (The y-axis is now linear.) Again, shorter episode lengths are better as they indicate that the agents are able to capture the rabbits in less steps. Note that for an optimal policy, any starting state will take less than 5 steps to complete (the farthest two points on the board).

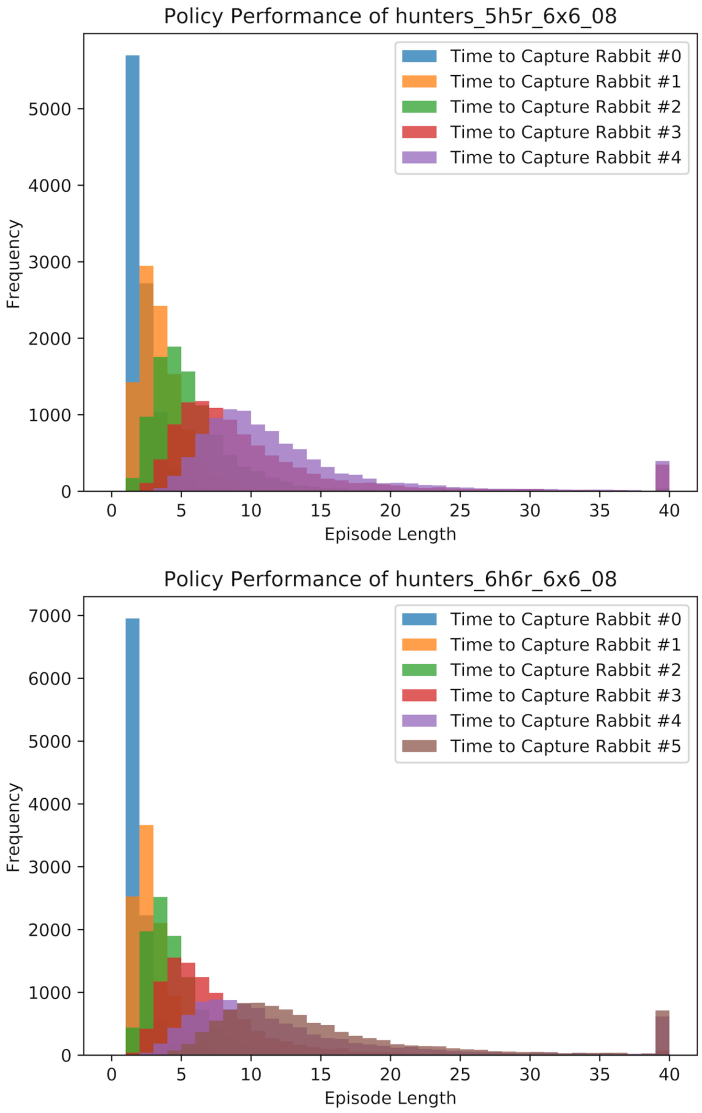
For 2 hunters, the multi-agent approach slightly outperforms the baseline as is it more left-skewed. For 3 hunters, the baseline significantly underperforms against the LSTM policy, having visibly deteriorated when scaling from 2 to 3 agents. The LSTM policy does not suffer such a large hit in performance. We tested this further by training on the game with 4 hunters, which at $9^4 = 5661$ joint actions, was too large to practically train a baseline on (given the resources available).



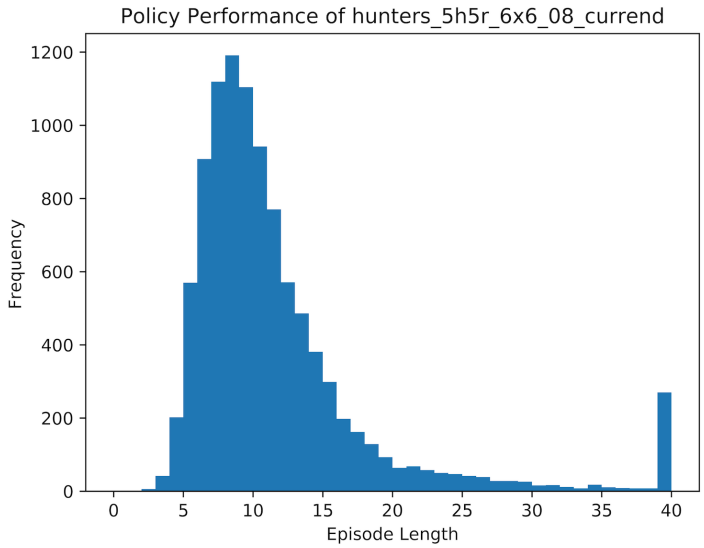
As the policy histogram shows, the LSTM policy still performs very well when increasing to 4 agents. It is almost always able to capture all 4 rabbits within 6 steps. These results are very promising, as the LSTM can solve a task that is too large for the baseline. However, performance deteriorates rapidly when increasing the number of agents to 5-6 hunters. Here are the policy performances for these two games,



It's unclear why there is a very sudden drop in performance, as almost no games took more than 10 steps to finish for 4 hunters, but this is the majority for 5 and 6 hunters. We can also look at the time-to-capture for each rabbit,



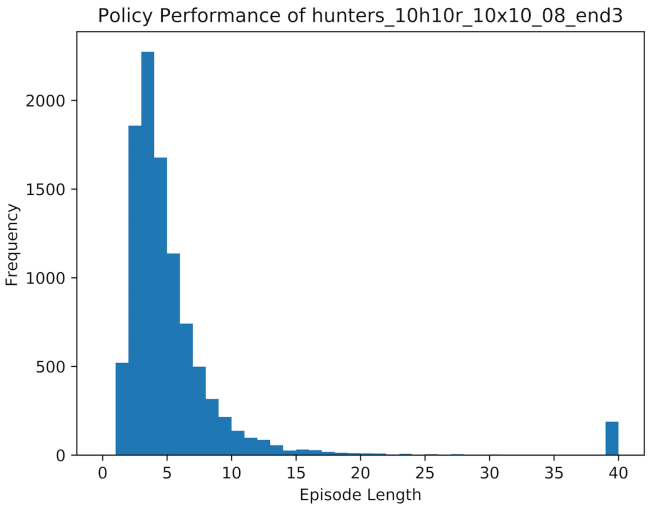
This shows that rabbits are generally steadily caught one after the other, but with a long tail of rabbits that take a longer time to capture. Next, we also tried a curriculum learning approach[1] where the network first trains for 100,000 episodes on a game that terminates early when just 3 rabbits are caught, then 4 rabbits, and so on. This would allow the agents to effectively learn simpler tasks that can serve as a better starting point for increasingly more complex tasks. However, this approach only had a very minor boost in performance,



A variation of the curriculum was also explored, where the game starts with only 3 hunters and rabbits, and an additional hunter and rabbit was added for every 100,000 episodes of training. The result of this was slightly worse than the former curriculum, so it was also unsuccessful. More investigation is needed to understand the drop in performance at 5 agents.

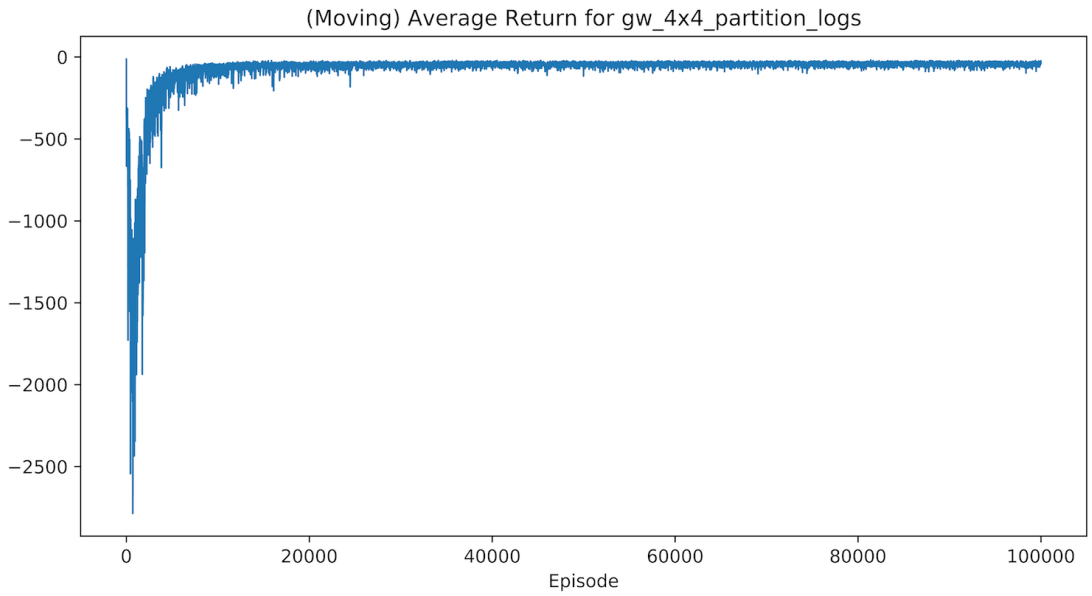
Reformulating the task, however, can provide an isolated demonstration of the multi-agent approach's ability to scale. If we terminate the hunters game early when just 3 rabbits are caught, regardless of the number of hunters or rabbits, our technique works very well. Though this game is simpler, it is still a task that the traditional single-agent approach can-

not tackle at all, as the joint action space is much too large for practical purposes. Examining the performance of 10 hunters and 10 rabbits on a 10×10 grid in this environment,



The LSTM learns a very good policy for this truncated task, and very quickly. Very few episodes take more than 9 time-steps to complete, which is the longest possible episode for an optimal policy.

Next, we can look at the SARSA approach. This was unsuccessful in finding a good policy. Here is a graph of its training performance for a 4×4 Gridworld:



There's a nice learning curve, indicating that SARSA is able to effectively learn at the beginning, but it gets trapped in a very suboptimal policy fairly quickly. The policy it converges to takes about 40 steps to reach the exit, which is ridiculous for a 4×4 grid.

To try to debug this, we implemented a target network which serves only as a stationary target for updating the policy network parameters, as in the DQN paper[5]. However, this proved to be ineffective and SARSA remained unsuccessful. It will also take more investigation to determine why this is occurring (perhaps a subtle bug in the code). However, since multi-agent SARSA is structurally very similar to multi-agent policy gradient, it may not provide better performance even when working.

Last, we tested various modifications for both policy gradient and SARSA, such as using an entropy term to encourage exploration[4], utilizing $\mathrm{TD}(k)$ returns rather than Monte-Carlo returns for training both the policy and value networks, various learning rates and discount factors, and others. However, a majority of these either did not affect or harmed performance, and the best combination of settings was used for the evaluation.

Conclusion

In this project, we have attempted to develop reinforcement learning algorithms that scale well to multiple agents, as opposed to traditional approaches where the network size is exponential in the number of agents. These algorithms lead to promising early results, which outperform the single-agent baseline for our Hunters task and even work on tasks too large for the baseline.

However, much future work remains. The most important issues include investigating why multi-agent policy gradient does not work for more than 5 hunters, and debugging our SARSA algorithm to escape from the local optima. We can also tune the hyperparameters for our algorithms further, such as the entropy, learning rate, choice of optimizer, using TD(k) returns, and others. We can also look at using the state-of-the-art DQNs and A3C as baselines that may perform better than our current baseline for a fairer comparison. Another point of comparison may be communication-based multi-agent systems[9]. Additionally, we may want to investigate other multi-agent environments with more dense rewards, as our current tasks only release sparse rewards, which may pose separate challenges and obstacles. Last, other approaches based on H-functions[6] or modifying the MDP were not implemented due to time constraints, but could be a promising avenue of exploration.

References

[1] Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning* (pp. 41–48).: ACM.

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[3] Karpathy, A. (2016). Deep reinforcement learning: Pong from pixels.

[4] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533.

[6] Pazis, J. & Parr, R. (2011). Generalized value functions for large action sets. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 1185–1192).

[7] Silver, D. (2015). Policy gradient methods.

[8] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587), 484–489.

[9] Sukhbaatar, S., Fergus, R., et al. (2016). Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems* (pp. 2244–2252).

[10] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

[11] Sutton, R. S. & Barto, A. G. (2017). *Reinforcement Learning: An Introduction (Draft)*.

[12] Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y., et al. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99 (pp. 1057–1063).

[13] Vieira, T. (2014). Exp-normalize trick.

[14] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4), 229–256.